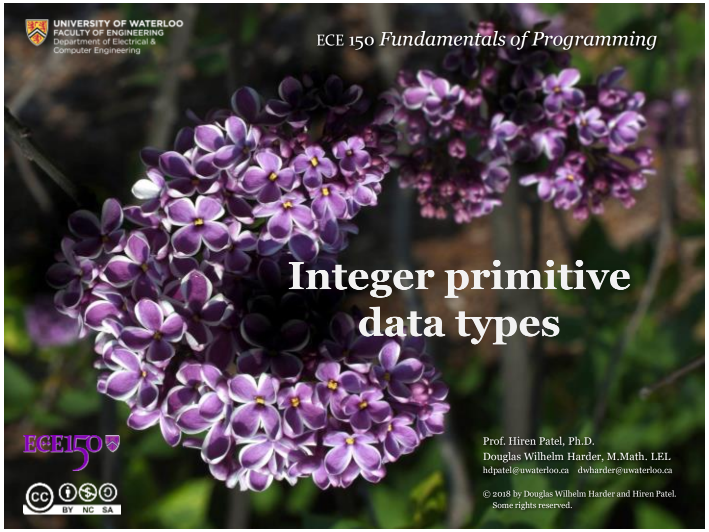
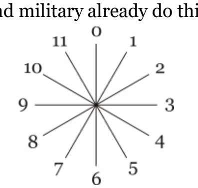


Outline

- In this lesson, we will:
  - Learn the representation of unsigned integers
  - Describe how integer addition and subtraction is performed
    - This requires the 2's complement representation
  - Use 2's complement to store negative numbers for signed integers
  - Describe the ranges stored by the four integer types
    - Both unsigned and signed

Clock arithmetic

- Suppose we have a clock face, but define 12 o'clock as "0" o'clock
  - The Europeans and military already do this…



- You know that:
  - 5 hours after 9 o'clock is 2 o'clock
  - 7 hours before 3 o'clock is 8 o'clock
  - Specifically:
    - 1 hour before 0 o'clock is 11 o'clock
    - 1 hour after 11 o'clock is 0 o'clock
- This is arithmetic modulo 12

`int` and `long`

- We have seen integer data types up to this point:

```
int
unsigned int
long
unsigned long
```

- It has been suggested that
  - An unsigned integer stores only positive numbers (`0`, `1`, `2`, …)
  - A `long` can store more information than an `int`
- We will now see how integers are stored in the computer

Binary representations

- We have already described binary numbers
  - On the computer, all integers are stored in binary
  - Thus, to store each of these numbers, we must store the corresponding binary digits (bits):

3	11	2
42	101010	6
616	1001101000	10
299792458	10001110111100111100001001010	29

  - To store a googol (10^{100}), we must store 333 bits:

```
10010010010011010110100100101100101001100001101111100111…
01011000001011001001111000010011000100110011100000010111111…
10011100010101100111001000000100011100010000100011010011…
11100101010101011001001000011000010001010100000101110100…
01111000100000000000000000000000000000000000000000000000…
0000000000000000000000000000000000000000000000000000
```


Storage

- Do we store as many bits as are necessary?
  - You could, but this would be exceedingly difficult to manage
- Instead, each primitive data type has a fixed amount of storage
  - 8 bits are defined as 1 byte
  - All data types are an integral number of bytes
    - Usually 1, 2, 4, 8 or 16 bytes
    - Because we use binary, powers of 2 are very common:

Exponent	Decimal	Binary
2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	10000
2^5	32	100000
2^6	64	1000000

`unsigned int`

- A variable is declared `unsigned int` is allocated four bytes
  - 4 bytes is 4 × 8 = 32 bits
  - 32 different `1`s and `0`s can be stored
  - The smallest and largest:

    `00000000000000000000000000000000`

    `11111111111111111111111111111111`
  - The smallest represents 0
  - The largest is one less than

$$1\underbrace{00000000000000000000000000000000}_{\text{32 zeros}}$$

  - This equals 2^{32}, thus, the largest value that can be stored as an `unsigned int` is $2^{32} - 1 = 4294967295$
    - Approximately 4 billion

`unsigned short`

- Sometimes, you don't need to store numbers this large
- Variables declared `unsigned short` are allocated two bytes
  - 2 bytes is 2 × 8 = 16 bits
  - 16 different `1`s and `0`s can be stored
  - The smallest and largest:

    `0000000000000000`

    `1111111111111111`
  - The smallest represents 0
  - The largest is one less than

$$1\underbrace{0000000000000000}_{\text{16 zeros}}$$

  - This equals 2^{16}, thus, the largest value that can be stored as an `unsigned int` is $2^{16} - 1 = 65535$

`unsigned long`

- Sometimes, you need to store very large numbers
- Variables declared `unsigned long` are allocated eight bytes
  - 8 bytes is 8 × 8 = 64 bits
  - 64 different `1`s and `0`s can be stored
  - The smallest and largest:

    `0000000000000000000000000000000000000000000000000000000000000000`
    `1111111111111111111111111111111111111111111111111111111111111111`
  - The smallest represents 0
  - The largest is one less than

    $1\,\underbrace{0000000000000000000000000000000000000000000000000000000000000000}_{\text{64 zeros}}$
  - This equals 2^{64}, thus, the largest value that can be stored as an `unsigned int` is $2^{64} - 1$ = 18446744073709551615
    - This 18 billion billion or 18 quintillion

Example

- Consider this program:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    unsigned short a{42};
    unsigned int   b{207500};
    unsigned long  c{299792458};

    std::cout << (a + b + c) << std::endl;

    return 0;
}
```

Note:
First, a is upcast to `unsigned int` before to the first addition
Then, the result is upcast to `unsigned long` before the second addition

Output:
`300000000`

Example

- On the stack, an appropriate number of bytes are allocated to each variable

$\underbrace{\texttt{??}}_{\text{8 bytes for c}}\underbrace{\texttt{????????????????????????????????}}_{\text{4 bytes for b}}\underbrace{\texttt{????????????????}}_{\text{2 bytes for a}}$

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    unsigned short a{42};
    unsigned int   b{207500};
    unsigned long  c{299792458};

    std::cout << (a + b + c) << std::endl;

    return 0;
}
```


Example

- Each of these variables is then initialized

$\underbrace{\texttt{0000000000000000000000000000000000010001110111100111100001001010}}_{\text{8 bytes for c}}\underbrace{\texttt{00000000000000110010101010001100}}_{\text{4 bytes for b}}\underbrace{\texttt{0000000000101010}}_{\text{2 bytes for a}}$

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    unsigned short a{42};
    unsigned int   b{207500};
    unsigned long  c{299792458};

    std::cout << (a + b + c) << std::endl;

    return 0;
}
```

Example

- Generally, however, we display the bytes in memory as a column of bytes, the values of which are concatenated

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    unsigned short a{42};
    unsigned int   b{207500};
    unsigned long  c{299792458};

    std::cout << (a + b + c) << std::endl;

    return 0;
}
```

8 bytes for c	00000000
	00000000
	00000000
	00000000
	00010001
	11011110
	01111000
	01001010
4 bytes for b	00000000
	00000011
	00101010
	10001100
2 bytes for a	00000000
	00101010

Wasted space?

- If an integer does not use all the bytes, the remaining bits are never-the-less allocated until the variable goes out of scope
  - In general-purpose computing, this is often not a problem
  - This is a critical issue, however, in embedded systems
    - More memory:
      - Costs more
      - Uses more power
      - Produces more heat

Determining the *size* of a type

- We have said `short`, `int` and `long` are 2, 4 and 8 bytes
  - This is true on most every general-purpose computer
- Unfortunately, the C++ specification doesn't require this
  - Fortunately, the `sizeof` operator gives you this information

```
#include <iostream>
int main();
int main() {
    std::cout << "An 'unsigned short' occupies "
              << sizeof ( unsigned short ) << " bytes" << std::endl;
    std::cout << "An 'unsigned int'   occupies "
              << sizeof ( unsigned int ) << " bytes" << std::endl;
    std::cout << "An 'unsigned long'  occupies "
              << sizeof ( unsigned long ) << " bytes" << std::endl;
    return 0;
}
```

Output on *ecelinux*:

```
An 'unsigned short' occupies 2 bytes
An 'unsigned int'   occupies 4 bytes
An 'unsigned long'  occupies 8 bytes
```


Memory and initial values

- Question:
  - What happens if the initial value cannot be stored?

```
#include <iostream>

int main();

int main() {
    unsigned short c{299792458};
    std::cout << "The speed of light is " << c
              << " m/s." << std::endl;

    return 0;
}
```

Memory and initial values

- Fortunately, you get a warning:

```
example.cpp: In function 'int main()':
example.cpp:6:31: warning: narrowing conversion of â299792458â from 'int' to
'short unsigned int' inside { } [-Wnarrowing]
     unsigned short c{299792458};
                               ^
example.cpp:6:31: warning: large integer implicitly truncated to unsigned
type [-Woverflow]
```

- It still compiles and executes:

```
The speed of light is 30794 m/s.
```


Memory and initial values

- Where does `30794` come from?

$$\overbrace{\texttt{00010001\,11011110\,}\underbrace{\texttt{01111000\,01001010}}_{\text{Only 16 bits are allocated}}}^{\texttt{c}\text{ requires 29 bits}}$$

- The binary number `0b111100001001010` equals 30794 in base 10

Memory and initial values

- Important:

All unsigned integers are stored:

modulo 2^{16} *for* `unsigned short`

modulo 2^{32} *for* `unsigned int`

modulo 2^{64} *for* `unsigned long`

Memory and arithmetic

- What happens if the sum, difference or product of two integers exceeds what can be stored?

```
#include <iostream>

int main();

int main() {
    unsigned short m1{40000}, m2{42000};
    int    n1{40000}, n2{42000};
    unsigned short sum{m1 + m2}, diff{m1 - m2}, prod{m1*m2};

    std::cout << sum  << "\t" << (n1 + n2) << std::endl;
    std::cout << diff << "\t" << (n1 - n2) << std::endl;
    std::cout << prod << "\t" << (n1*n2)   << std::endl;

    return 0;
}
```

Output:

```
16464    82000
63536    -2000
50176    1680000000
```

Memory and arithmetic

- Let's look at the actual values and the evaluated results:

16464	0100000001010000
82000	10100000001010000
63536	1111100000110000
–2000	-0000011111010000
50176	1100010000000000
1680000000	1100100001000101100010000000000

- For the sum and product, the result ignores the higher-order bits
  - The negative number is a little odd....

Memory and arithmetic

- What happens if the sum, difference or product of two integers exceeds what can be stored?

```
#include <iostream>
int main();
int main() {
    unsigned short smallest{0}, largest{65535};

    std::cout << "Smallest:  " << smallest << std::endl;
    std::cout << "Largest:   " << largest  << std::endl;
    --smallest;
    ++ largest;
    std::cout << "Smallest minus 1: " << smallest << std::endl;
    std::cout << "Largest plus 1:   " << largest  << std::endl;

    return 0;
}
```

Output:

```
Smallest:  0
Largest:   65535
Smallest minus 1: 65535
Largest plus 1:   0
```


Memory and arithmetic

- Important:

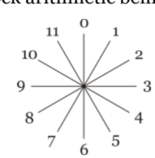
  All unsigned integers arithmetic is performed:

  modulo 2^{16} *for* `unsigned short`

  modulo 2^{32} *for* `unsigned int`

  modulo 2^{64} *for* `unsigned long`

- This is similar to all clock arithmetic being performed modulo 12



Addition

- Addition is easy:
  - Like in elementary school, line them up and occasionally you require a carry in the next column:
  - The rules are:
    - `0 + 0 → 0`
    - `0 + 1 → 1`
    - `1 + 1 → 10 → 0` with a carry of 1
    - `1 + 1 + 1 → 11 → 1` with a carry of 1
  - For example, adding two `unsigned short`:

```
  1   1111    1
  0101000111010110     20950
+ 1001101011000100     39620
  1110110010011010     60570
```

Addition

- What if we go over? Adding these two `unsigned short`:

```
 1   1   1111    1          53718
 1101000111010110
+1001101011000100           39620
10110110010011010           93338
```

- The additional bit is discarded—addition is calculated modulo 2^{16}
  - Thus, the answer is `110110010011010` which is 27802

Subtraction

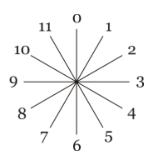
- Subtraction is more difficult:
  - Like in elementary school, you learned to “borrow”, but borrowing may require you to look way ahead:

```
  0100000001010000
- 0001101011000101
         ?
```

  - Our salvation: we are performing arithmetic modulo 65536

Subtraction

- Going back to the clock:
  - Subtracting 10 is the same as adding 2
  - Subtracting 4 is the same as adding 8
  - Subtracting 9 is the same as adding 3
- Thus, to subtract n, add $12 - n$



- In our case, to subtract n, add $65536 - n$

```
                                  1 1       1 1 1
  0100000001010000  ← 16464 →    0100000001010000
- 0001101011000101  6853       + 1110010100111011   58683
         ?                      10010010110001011
```

  - The answer is `0010010110001011` — 9611

Subtraction

- The million-dollar question:

  How do you calculate $65536 - n$???

- Subtract any number from 9999999999999, no borrows are needed

```
  9999999999999
– 5501496383498
  4498503616501
```

- Thus, to calculate $10000000000000 - n$, instead calculate

  $(10000000000000 - 1) - n + 1 = (9999999999999 - n) + 1$

- For example:

```
  10000000000000
– 5501496383498
  4498503616502
```

This is called the base-10 complement or “10’s complement”
– this is how older adding machines performed subtraction

Subtraction

- In binary, the equivalent is base-2 complement or "2's complement"
  - To calculate 65536 – 1970, calculate (65535 – 1970) + 1:

```
  1111111111111111
– 0000011110110010
  1111100001001101
+                1
  1111100001001110
```

- Thus, to calculate 2018 – 1970, just add the 2's complement of 1970 to 2018:

```
  0000011111100010
+ 1111100001001110
```

~~1~~0000000000110000

- This is the binary representation of $48 = 2^5 + 2^4 = 32 + 16$
  - Remember, we ignore the leading 1

2's complement

- To calculate the 2's complement:
  1. Complement all of the bits in the number
     - This includes leading zeros
  2. Add 1
- For example, the 2's complement of the speed of light is stored as an `unsigned int` is

```
  00010001110111110011110000001001010
  11101110001000001100001111011010 1
+                                1
  11101110001000001100001111011 0110
```


2's complement

- There is a faster way to compute it without the addition:
  - Scan from right-to-left
    - Find the first 1, and then flip each bit to the left of that
- The 2's complement of each of the following is given below it

```
1011011111011111
0100100000100001

1010111111100000
0101000000100000

0000100100101100
1111011011010100
```


2's complement

- The 2's complement of `0` stored as an `unsigned int` is

```
  00000000000000000000000000000000
  11111111111111111111111111111111
+                                1
```

~~1~~00000000000000000000000000000000

- This makes sense: any number minus zero is unchanged

2's complement

- The 2's complement algorithm is self-inverting:
  - If n is a number, then $2^{16} - (2^{16} - n) = n$
  - The 2's complement of the 2's complement of a number is the number itself

```
  1110110010111110
  0001001101000001
+ ________________1
  0001001101000010
  1110110010111101
+ ________________1
  1110110010111110
```

- That is, $f^{-1} = f$ or $f(f(n)) = n$

Memory and arithmetic

- Try it yourself:

```
#include <iostream>

int main();

//////////////////////////////////////////////////
// Verify that every possible sum, difference,
// product, division and remainder option on
// 'unsigned short' is the actual operation
// modulo 65536
//////////////////////////////////////////////////

int main() {
   long const TWO_15{32768};
   long const TWO_16{2*TWO_15};

   for ( long i{0}; i < TWO_16; ++i ) {
      // Just print out the count when we get to multiples of 1024
      if ( (i % 1024) == 0 ) {
         std::cout << i << std::endl;
      }

      for ( long j{0}; j < TWO_16; ++j ) {
         unsigned short si{i};
         unsigned short sj{j};

         // Perform the sum as unsigned shorts and as signed longs
         unsigned short sk{si + sj};
         long k{(i + j) % TWO_16};

         if ( k < 0 ) {
            k += TWO_16;
         }

         // If they differ, print out a warning
         if ( k != sk ) {
            std::cout << i << " + " << j << " = " << k
                      << " != " << sk << std::endl;
         }

         // Perform the difference as unsigned shorts and as signed longs
         sk = si - sj;
         k = (i - j) % TWO_16;

         if ( k < 0 ) {
            k += TWO_16;
         }

         // If they differ, print out a warning
         if ( k != sk ) {
            std::cout << i << " - " << j << " = " << k
                      << " != " << sk << std::endl;
         }

         // Perform the difference as unsigned shorts and as signed longs
         sk = si * sj;
         k = (i * j) % TWO_16;

         if ( k < 0 ) {
            k += TWO_16;
         }


         // If they differ, print out a warning
         if ( k != sk ) {
            std::cout << i << " * " << j << " = " << k
                      << " != " << sk << std::endl;
         }

         // We cannot perform division or remainder when the
         // second operand is zero, so skip these operations
         if ( j != 0 ) {
            // Perform the division as unsigned shorts and as signed longs
            sk = si / sj;
            k = (i / j) % TWO_16;

            if ( k < 0 ) {
               k += TWO_16;
            }

            // If they differ, print out a warning
            if ( k != sk ) {
               std::cout << i << " / " << j << " = " << k
                         << " != " << sk << std::endl;
            }

            // Perform the remainder as unsigned shorts and as signed longs
            sk = si % sj;
            k = (i % j) % TWO_16;

            if ( k < 0 ) {
               k += TWO_16;
            }

            // If they differ, print out a warning
            if ( k != sk ) {
               std::cout << i << " % " << j << " = " << k
                         << " != " << sk << std::endl;
            }
         }
      }
   }

   return 0;
}
```


Summary so far

- We have the following:
  - Unsigned integers are stored as either 1, 2, 4 or 8 bytes
  - The value is stored in the binary representation

Type	Bytes	Bits	Range	Approximate Range
`unsigned char`	1	8	$0, \ldots, 2^8 - 1$	0, …, 255
`unsigned short`	2	16	$0, \ldots, 2^{16} - 1$	0, …, 65535
`unsigned int`	4	32	$0, \ldots, 2^{32} - 1$	0, …, 4.3 billion
`unsigned long`	8	64	$0, \ldots, 2^{64} - 1$	0, …, 18 quintillion

  - You should not memorize the exact ranges

Useful tool…

- Note that $2^{10} = 1024$, so $2^{10} \approx 1000 = 10^3$
  - We can use this to estimate magnitudes:
    - $2^{12} = 2^2\, 2^{10} \approx 4 \times 1000 = 4000$
    - $2^{16} = 2^6\, 2^{10} \approx 64 \times 1000 = 64000$
    - $2^{24} = 2^4\, 2^{20} = 2^4\, (2^{10})^2 \approx 16 \times 1000^2 = 16$ million
    - $2^{32} = 2^2\, 2^{30} = 2^2\, (2^{10})^3 \approx 4 \times 1000^3 = 4$ billion
  - This approximation will underestimate by approximately 2%

Signed types

- We've seen that `short`, `int` and `long` all allows you to store both positive and negative integers
  - How do we store such negative numbers?
- Because we have two choices (positive or negative), we could use one bit to represent the *sign*: `0` for positive, `1` for negative
  - For example:

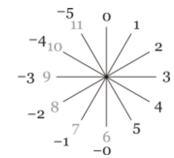
The *sign* bit

32767	`0111111111111111`
2	`0000000000000010`
1	`0000000000000001`
0	`0000000000000000`
–0 ?	`1000000000000000`
–1	`1000000000000001`
–2	`1000000000000010`
–32768	`1111111111111111`

–0 = 0, so do we have two zeros?

Signed types

- This is similar to marking the hours of a clock as follows:

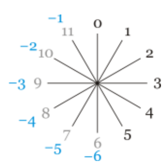


- Unfortunately, this leads to ugly arithmetic operations…

  $-1 + 1 = 0$ or -0, but $7 + 1 = 8$
  $-5 + 2 = -3$, but $11 + 2 = 1$

Signed types

- A better solution:



- Note that

  $-1 + 1 = 0$, but also $11 + 1 = 0$
  $-5 + 2 = -3$, but also $7 + 2 = 9$, which we are equating to -3

Signed integers

- Here is a workable solution:
  - If the leading bit is `0`:
    - Assume the remainder of the number is the integer represented
    - For `short`, this includes

      `0000000000000000` 0
      `0111111111111111` $2^{15} - 1 = 32767$
    - This includes 2^{15} different positive numbers
  - If the leading bit is `1`:
    - Assume the number is negative and its magnitude can be found by applying the 2's complement algorithm
    - Recall the 2's complement algorithm is self-inverting

Signed integers

- For negative numbers stored as a `short`:

```
  1000000000000000
  0111111111111111
+                1
  1000000000000000
```

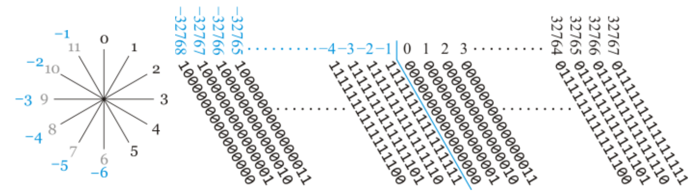
  - This is the representation of the largest negative number: -2^{15}

```
  1111111111111111
  0000000000000000
+                1
  0000000000000000
```

  - This is the representation of the smallest negative number: –1

Signed integers

- Here, you can compare these two techniques
  - In both cases, we go from –12/2 to 12/2 – 1 and $-2^{16}/2$ to $2^{16}/2 - 1$



Signed integers

- For example, `1111111111010110` is a negative `short`

```
  1111111111010110
  0000000000101001
+                1
  0000000000101010
```

- Thus, it represents –42
- Let's calculate –42 + 91 = 49 and –42 – 91 = –133:

```
  1111111111010110
+ 0000000001011011
 10000000000110001   ← 49

  1111111111010110
+ 1111111110100101   ← –91
 11111111101111011   ← –133      10000101 ← 133
```


Summary

- To summarize:
  - Integer types are stored as either 1, 2, 4 or 8 bytes
  - Negative numbers are stored in the 2's complement representation

Type	Bytes	Bits	Range	Approximate Range
`unsigned char`	1	8	$0, \ldots, 2^{8} - 1$	0, …, 255
`unsigned short`	2	16	$0, \ldots, 2^{16} - 1$	0, …, 65535
`unsigned int`	4	32	$0, \ldots, 2^{32} - 1$	0, …, 4.3 billion
`unsigned long`	8	64	$0, \ldots, 2^{64} - 1$	0, …, 18 quintillion
`signed char`	1	8	$-2^{7}, \ldots, 2^{7} - 1$	–128, …, 127
`short`	2	16	$-2^{15}, \ldots, 2^{15} - 1$	–32768, …, 32767
`int`	4	32	$-2^{31}, \ldots, 2^{31} - 1$	–2.15 billion, …, 2.15 billion
`long`	8	64	$-2^{63}, \ldots, 2^{63} - 1$	–9 quintillion, …, 9 quintillion

Warning

- While common, the C++ standard does not require these sizes:
  - Each compiler may choose sizes so long as the following are true:

```
assert( sizeof( char ) == 1 );
assert( sizeof( short ) >= 2 );      // At least 16 bits
assert( sizeof( int ) >= sizeof( short ) );
     // At least as large as 'short'
assert( sizeof( long ) >= 4 );       // At least 32 bits
assert( sizeof( long long ) >= 8 );  // At least 64 bits
```

- In GNU g++, the sizes are as we have described in this slide deck
- In Microsoft Visual Studio, however:
  - A `long` is only four bytes (same as `int`)
  - A `long long` is eight bytes
  - We do not use `long long` in this course
    - You may have to use it if you program in Visual Studio

Summary

- Following this lesson, you now
  - Understand the representation of unsigned integers
  - Know how to perform subtraction using 2's complement
    - Similar to 10's complement used a century ago
  - Understand that signed integers store negative numbers in their 2's complement representation
  - Know that `char` is actually just an integer type
    - It can be interpreted as a printable character if necessary
  - Understand the ranges stored by `char`, `short`, `int` and `long`

References

[1] Wikipedia
https://en.wikipedia.org/wiki/Integer_(computer_science)
https://en.wikipedia.org/wiki/Two%27s_complement

Acknowledgments

Theresa DeCola and Charlie Liu.

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer